

Contents

1 Setting	1
1.1 vimrc	1
2 Math	1
2.1 Basic Arithmetic	1
2.2 Sieve Methods : Prime, Divisor, Euler phi	2
2.3 Primality Test	2
2.4 Chinese Remainder Theorem	3
2.5 Burnside's Lemma	3
2.6 Kirchoff's Theorem	3
2.7 Fast Fourier Transform	3
2.8 Matrix Operations	3
2.9 Gaussian Elimination	3
2.10 Simplex Algorithm	3
3 Data Structure	3
3.1 Order statistic tree	3
3.2 Fenwick Tree	3
3.3 Segment Tree with Lazy Propagation	4
3.4 Persistent Segment Tree	4
3.5 Link/Cut Tree	4
4 DP	4
4.1 Convex Hull Optimization	4
4.2 Divide & Conquer Optimization	4
4.3 Knuth Optimization	4
5 Graph	5
5.1 SCC (Tarjan)	5
5.2 SCC (Kosaraju)	5
5.3 2-SAT	5
5.4 BCC, Cut vertex, Bridge	5
5.5 Heavy-Light Decomposition	5
5.6 Bipartite Matching (Hopcroft-Karp)	5
5.7 Maximum Flow (Edmonds-Karp)	5
5.8 Maximum Flow (Dinic)	5
5.9 Min-cost Maximum Flow	5
6 Geometry	5
6.1 Basic Operations	5
6.2 Convex Hull	5
6.3 Polygon Cut	5
6.4 Compare angles	5

7 String	5
7.1 KMP	5
7.2 Aho-Corasick	5
7.3 Suffix Array with LCP	5
7.4 Suffix Tree	5
7.5 Manacher's Algorithm	5
8 Miscellaneous	5
8.1 Fast I/O	5
8.2 Magic Numbers	5

1 Setting

1.1 vimrc

```
set ts=4 sts=4 sw=4
set ai si nu
```

2 Math

2.1 Basic Arithmetic

```
typedef long long ll;
typedef unsigned long long ull;

// calculate ceil(a/b)
// |a|, |b| <= (2^63)-1 (does not cover -2^63)
ll ceildiv(ll a, ll b) {
    if (b < 0) return ceildiv(-a, -b);
    if (a < 0) return (-a) / b;
    return ((ull)a + (ull)b - 1ull) / b;
}

// calculate floor(a/b)
// |a|, |b| <= (2^63)-1 (does not cover -2^63)
ll floordiv(ll a, ll b) {
    if (b < 0) return floordiv(-a, -b);
    if (a >= 0) return a / b;
    return -(ll)((ull)(-a) + b - 1) / b;
}

// calculate a*b % m
// x86-64 only
ll large_mod_mul(ll a, ll b, ll m)
{
    return ll((__int128)a*(__int128)b%m);
}
```

```

// calculate a*b % m
// |m| < 2^62, x86 available
// O(logb)
ll large_mod_mul(ll a, ll b, ll m)
{
    a %= m; b %= m; ll r = 0, v = a;
    while (b) {
        if (b&1) r = (r + v) % m;
        b >>= 1;
        v = (v << 1) % m;
    }
    return r;
}

// calculate n^k % m
ll modpow(ll n, ll k, ll m) {
    ll ret = 1;
    n %= m;
    while (k) {
        if (k & 1) ret = large_mod_mul(ret, n, m);
        n = large_mod_mul(n, n, m);
        k /= 2;
    }
    return ret;
}

// calculate gcd(a, b)
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

// find a pair (c, d) s.t. ac + bd = gcd(a, b)
pair<ll, ll> extended_gcd(ll a, ll b) {
    if (b == 0) return { 1, 0 };
    auto t = extended_gcd(b, a % b);
    return { t.second, t.first - t.second * (a / b) };
}

// find x in [0,m) s.t. ax == gcd(a, m) (mod m)
ll modinverse(ll a, ll m) {
    return (extended_gcd(a, m).first % m + m) % m;
}

// calculate modular inverse for 1 ~ n
void calc_range_modinv(int n, int mod, int ret[]) {
    ret[1] = 1;
    for (int i = 2; i <= n; ++i)
        ret[i] = (ll)(mod - mod/i) * ret[mod%i] % mod;
}

```

2.2 Sieve Methods : Prime, Divisor, Euler phi

```

// find prime numbers in 1 ~ n
// ret[x] = false -> x is prime
// O(n*loglogn)

```

```

void sieve(int n, bool ret[]) {
    for (int i = 2; i * i <= n; ++i)
        if (!ret[i])
            for (int j = i * i; j <= n; j += i)
                ret[j] = true;
}

// calculate number of divisors for 1 ~ n
// when you need to calculate sum, change += 1 to += i
// O(n*logn)
void num_of_divisors(int n, int ret[]) {
    for (int i = 1; i <= n; ++i)
        for (int j = i; j <= n; j += i)
            ret[j] += 1;
}

// calculate euler totient function for 1 ~ n
// phi(n) = number of x s.t. 0 < x < n && gcd(n, x) = 1
// O(n*loglogn)
void euler_phi(int n, int ret[]) {
    for (int i = 1; i <= n; ++i) ret[i] = i;
    for (int i = 2; i <= n; ++i)
        if (ret[i] == i)
            for (int j = i; j <= n; j += i)
                ret[j] -= ret[j] / i;
}

```

2.3 Primality Test

```

bool test_witness(ull a, ull n, ull s) {
    if (a >= n) a %= n;
    if (a <= 1) return true;
    ull d = n >> s;
    ull x = modpow(a, d, n);
    if (x == 1 || x == n-1) return true;
    while (s-- > 1) {
        x = large_mod_mul(x, x, n);
        x = x * x % n;
        if (x == 1) return false;
        if (x == n-1) return true;
    }
    return false;
}

// test whether n is prime
// based on miller-rabin test
// O(logn*logn)
bool is_prime(ull n) {
    if (n == 2) return true;
    if (n < 2 || n % 2 == 0) return false;

    ull d = n >> 1, s = 1;
    for(;; (d&1) == 0; s++) d >>= 1;

#define T(a) test_witness(a##ull, n, s)

```

```

    if (n < 4759123141ull) return T(2) && T(7) && T(61);
    return T(2) && T(325) && T(9375) && T(28178)
        && T(450775) && T(9780504) && T(1795265022);
#undef T
}

```

2.4 Chinese Remainder Theorem

```

// find x s.t.  x === a[0] (mod n[0])
//              === a[1] (mod n[1])
//              ...
// assumption: gcd(n[i], n[j]) = 1
ll chinese_remainder(ll* a, ll* n, int size) {
    if (size == 1) return *a;
    ll tmp = modinverse(n[0], n[1]);
    ll tmp2 = (tmp * (a[1] - a[0]) % n[1] + n[1]) % n[1];
    ll ora = a[1];
    ll tgcd = gcd(n[0], n[1]);
    a[1] = a[0] + n[0] / tgcd * tmp2;
    n[1] *= n[0] / tgcd;
    ll ret = chinese_remainder(a + 1, n + 1, size - 1);
    n[1] /= n[0] / tgcd;
    a[1] = ora;
    return ret;
}

```

2.5 Burnside's Lemma

경우의 수를 세는데, 특정 transform operation(회전, 반사, ..)해서 같은 경우들은 하나로 친다. 전체 경우의 수는?

- 각 operation마다 이 operation을 했을 때 변하지 않는 경우의 수를 센다 (단, “아무것도 하지 않는다”라는 operation도 있어야 함!)

- 전체 경우의 수를 더한 후, operation의 수로 나눈다. (답이 맞다면 항상 나누어 떨어져야 한다)

2.6 Kirchoff's Theorem

그래프의 스패닝 트리의 개수를 구하는 정리.

무향 그래프의 Laplacian matrix L 를 만든다. 이것은 (정점의 차수 대각 행렬) - (인접행렬)이다. L 에서 행과 열을 하나씩 제거한 것을 L' 라 하자. 어느 행/열이든 관계 없다. 그래프의 스패닝 트리의 개수는 $\det(L')$ 이다.

2.7 Fast Fourier Transform

2.8 Matrix Operations

2.9 Gaussian Elimination

2.10 Simplex Algorithm

3 Data Structure

3.1 Order statistic tree

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp>
#include <functional>
#include <iostream>
using namespace __gnu_pbds;
using namespace std;

// tree<key_type, value_type(set if null), comparator, ...>
using ordered_set = tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update>;

int main()
{
    ordered_set X;
    for (int i = 1; i < 10; i += 2) X.insert(i); // 1 3 5 7 9
    cout << boolalpha;
    cout << *X.find_by_order(2) << endl; // 5
    cout << *X.find_by_order(4) << endl; // 9
    cout << (X.end() == X.find_by_order(5)) << endl; // true

    cout << X.order_of_key(-1) << endl; // 0
    cout << X.order_of_key(1) << endl; // 0
    cout << X.order_of_key(4) << endl; // 2
    X.erase(3);
    cout << X.order_of_key(4) << endl; // 1
    for (int t : X) printf("%d ", t); // 1 5 7 9
}

```

3.2 Fenwick Tree

```

const int TSIZE = 100000;
int tree[TSIZE + 1];

// Returns the sum from index 1 to p, inclusive
int query(int p) {
    int ret = 0;
    for (; p > 0; p -= p & -p) ret += tree[p];
}

```

```

    return ret;
}

// Adds val to element with index pos
void add(int p, int val) {
    for (; p <= TSIZE; p += p & -p) tree[p] += val;
}

```

3.3 Segment Tree with Lazy Propagation

3.4 Persistent Segment Tree

3.5 Link/Cut Tree

4 DP

4.1 Convex Hull Optimization

$$O(n^2) \rightarrow O(n \log n)$$

조건 1) DP 점화식 꼴

$$D[i] = \min_{j < i} (D[j] + b[j] * a[i])$$

조건 2) $b[j] \leq b[j + 1]$

특수조건) $a[i] \leq a[i + 1]$ 도 만족하는 경우, 마지막 쿼리의 위치를 저장해두면 이분검색이 필요없어지기 때문에 amortized $O(n)$ 에 해결할 수 있음

4.2 Divide & Conquer Optimization

$$O(kn^2) \rightarrow O(kn \log n)$$

조건 1) DP 점화식 꼴

$$D[t][i] = \min_{j < i} (D[t - 1][j] + C[j][i])$$

조건 2) $A[t][i]$ 는 $D[t][i]$ 의 답이 되는 최소의 j 라 할 때, 아래의 부등식을 만족해야 함

$$A[t][i] \leq A[t][i + 1]$$

조건 2-1) 비용 C 가 다음의 사각부등식을 만족하는 경우도 조건 2)를 만족하게 됨

$$C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$$

4.3 Knuth Optimization

$$O(n^3) \rightarrow O(n^2)$$

조건 1) DP 점화식 꼴

$$D[i][j] = \min_{i < k < j} (D[i][k] + D[k][j]) + C[i][j]$$

조건 2) $A[i][j]$ 는 $D[i][j]$ 의 답이 되는 최소의 k 라 할 때, 아래의 부등식을 만족해야 함

$$A[i][j - 1] \leq A[i][j] \leq A[i + 1][j]$$

5 Graph

5.1 SCC (Tarjan)

5.2 SCC (Kosaraju)

5.3 2-SAT

5.4 BCC, Cut vertex, Bridge

5.5 Heavy-Light Decomposition

5.6 Bipartite Matching (Hopcroft-Karp)

5.7 Maximum Flow (Edmonds-Karp)

5.8 Maximum Flow (Dinic)

5.9 Min-cost Maximum Flow

6 Geometry

6.1 Basic Operations

6.2 Convex Hull

6.3 Polygon Cut

6.4 Compare angles

7 String

7.1 KMP

7.2 Aho-Corasick

7.3 Suffix Array with LCP

7.4 Suffix Tree

7.5 Manacher's Algorithm